3

PostgreSQL SQL Syntax and Use

The first two chapters explored the basics of the SQL language and looked at the data types supported by PostgreSQL. This chapter covers a variety of topics that should round out your knowledge of PostgreSQL.

We'll start by looking at the rules that you have to follow when choosing names for tables, columns, indexes, and such. Next, you'll see how to create, destroy, and view PostgreSQL databases. In Chapter 1, "Introduction to PostgreSQL and SQL," you created a few simple tables; in this chapter, you'll learn all the details of the `CREATE TABLE` command. I'll also talk about indexes. I'll finish up by talking about transaction processing and locking. If you are familiar with Sybase, DB2, or Microsoft SQL Server, I think you'll find that the locking model used by PostgreSQL is a refreshing change.

PostgreSQL Naming Rules

When you create an object in PostgreSQL, you give that object a name. Every table has a name, every column has a name, and so on. PostgreSQL uses a single data type to define all object names: the `name` type.

A value of type `name` is a string of 63 or fewer characters[1]. A `name` must start with a letter or an underscore; the rest of the string can contain letters, digits, and underscores.

If you examine the entry corresponding to `name` in the `pg_type` table, you will find that a `name` is really 64 characters long. Because the `name` type is used internally by the PostgreSQL engine, it is a null-terminated string. So, the maximum length of a `name` value is 63 characters. You can enter more than 63 characters for an object name, but PostgreSQL stores only the first 63 characters.

[1] You can increase the length of the name data type by changing the value of the NAMEDATALEN symbol before compiling PostgreSQL.

Both SQL and PostgreSQL reserve certain words and normally, you cannot use those words to name objects. Examples of reserved words are

```
ANALYZE
BETWEEN
CHARACTER
INTEGER
CREATE
```

You cannot create a table named `INTEGER` or a column named `BETWEEN`. A complete list of reserved words can be found in Appendix B of the *PostgreSQL User's Guide*.

If you find that you need to create an object that does not meet these rules, you can enclose the name in double quotes. Wrapping a name in quotes creates a quoted identifier. For example, you could create a table whose name is `"3.14159"`—the double quotes are required, but are not actually a part of the name (that is, they are not stored and do not count against the 63-character limit). When you create an object whose name must be quoted, you have to include the quotes not only when you create the object, but every time you refer to that object. For example, to select from the table mentioned previously, you would have to write

```
SELECT filling, topping, crust FROM "3.14159";
```

Here are a few examples of both valid and invalid names:

```
my_table       -- valid
my_2nd_table   -- valid
échéanciers    -- valid: accented and non-Latin letters are allowed
"2nd_table"    -- valid: quoted identifier
"create table" -- valid: quoted identifier
"1040Forms"    -- valid: quoted identifier
2nd_table      -- invalid: does not start with a letter or an underscore
```

Quoted names are case-sensitive. `"1040Forms"` and `"1040FORMS"` are two distinct names. Unquoted names are converted to lowercase, as shown here:

```
movies=# CREATE TABLE FOO( BAR INTEGER );
CREATE
movies=# CREATE TABLE foo( BAR INTEGER );
ERROR: Relation 'foo' already exists
movies=# \d
            List of relations
       Name       | Type  |     Owner
------------------+-------+---------------
 1040FORMS        | table | bruce
 1040Forms        | table | sheila
 customers        | table | bruce
 foo              | table | bruce
 rentals          | table | bruce
 tapes            | table | bruce
 (6 rows)
```

The names of all objects must be unique within some scope. Every database must have a unique name; the name of a schema must be unique within the scope of a single database, the name of a table must be unique within the scope of a single schema, and column names must be unique within a table. The name of an index must be unique within a database.

The Importance of the COMMENT Command

If you've been a programmer (or database developer) for more than, say, two days, you understand the importance of commenting your code. A comment helps new developers understand how your program (or database) is structured. It also helps *you* remember what you were thinking when you come back to work after a long weekend. If you're writing procedural code (in C, Java, PL/pgSQL, or whatever language you prefer), you can intersperse comments directly into your code. If you're creating objects in a PostgreSQL database, where do you store the comments? In the database, of course. The COMMENT command lets you associate a comment with just about any object that you can define in a PostgreSQL database. The syntax for the COMMENT command is very simple:

```
COMMENT ON object-type object-name IS comment-text;
```

where `object-type` and *`object-name`* are taken from the following:

```
DATABASE database-name
SCHEMA schema-name
TABLE table-name
COLUMN table-name.column-name
INDEX index-name
DOMAIN domain-name
TYPE data-type-name
VIEW view-name
CONSTRAINT constraint-name ON table-name
SEQUENCE sequence-name
TRIGGER trigger-name ON table-name
```

You can also define comments for other object types (functions, operators, rules, even languages), but the object types that we've shown here are the most common (see the PostgreSQL reference documentation for a complete list).

To add a comment to a table, for example, you would execute a command such as

```
COMMENT ON TABLE customers IS 'List of active customers';
```

You can only store one comment per object—if you COMMENT ON an object twice, the second comment replaces the first. To drop a comment, execute a COMMENT command, but specify NULL in place of the *`comment-text`* string, like this:

```
COMMENT ON TABLE customers IS NULL;
```

Once you have added a comment to an object, you can view the comment (in `psql`) using the command `\dd object-name-pattern`, like this:

```
movies=# \dd customers
                Object descriptions
 Schema |   Name    | Object |       Description
--------+-----------+--------+-------------------------
 public | customers | table  | List of active customers
(1 row)
```

The `\dd` command will show you any commented object whose name matches the *`object-name-pattern`*. The `\dd` command will *not* show comments that you've assigned to a column within a table. To see column-related comments, use the command `\d+ [table-name]`. To see the comment assigned to each database, use the command `\l+`.

Creating, Destroying, and Viewing Databases

Before you can do anything else with a PostgreSQL database, you must first create the database. Before you get too much further, it might be a good idea to see where a database fits into the overall scheme of PostgreSQL. Figure 3.1 shows the relationships between clusters, databases, schemas, and tables.

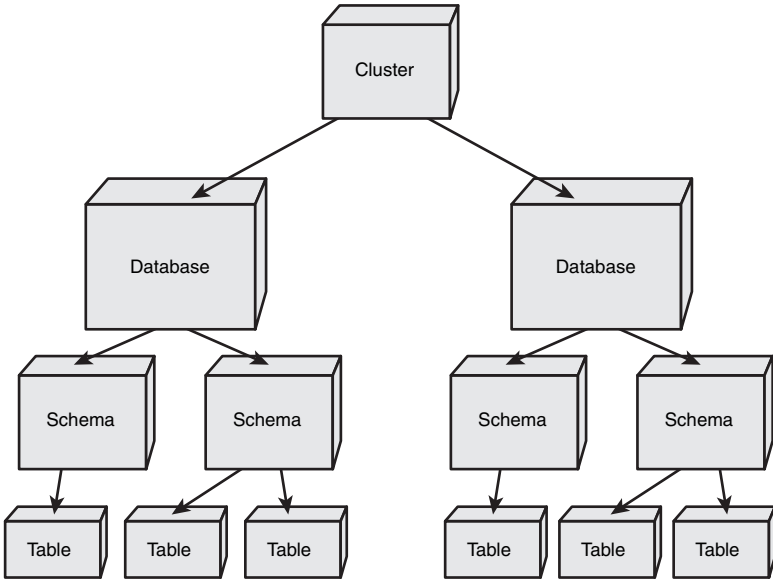

Figure 3.1 Clusters, databases, schemas, and tables.

At the highest level of the PostgreSQL storage hierarchy is the *cluster*. A cluster is a collection of databases. Each cluster exists within a single directory tree, and the entire cluster is serviced by a single `postmaster`. A cluster is not named—there is no way to refer to a cluster within PostgreSQL, other than by contacting the `postmaster` servicing that

cluster. The $PGDATA environment variable should point to the root of the cluster's directory tree. A cluster is serviced by a single postmaster process. The postmaster listens for connection requests coming from client applications. When a connection request is received (and the user's credentials are authenticated), the postmaster starts a new server process and connects the client to the server. A single client connection can only interact with a single database at any given time (but a client application can certainly open multiple connections if it needs to interact with several databases simultaneously). A postmaster process can connect a client application to any of the databases in the cluster serviced by that postmaster.

Four system tables are shared between all databases in a cluster: pg_group (the list of user groups), pg_database (the list of databases within the cluster), pg_shadow (the list of valid users), and pg_tablespace (the list of tablespaces).

Each cluster contains one or more databases. Every database has a name that must follow the naming rules described in the previous section. Database names must be unique within a cluster. A database is a collection of schemas.

A *schema* is a named collection of tables (as well as functions, data types, and operators). The schema name must be unique within a database. Table names, function names, index names, type names, and operators must be unique within the schema. A schema exists primarily to provide a naming context. You can refer to an object in any schema within a single database by prefixing the object name with schema-name. For example, if you have a schema named bruce, you can create a table within that schema as

```
CREATE TABLE bruce.ratings ( ... );
SELECT * FROM bruce.ratings;
```

Each connection has a schema search path. If the object that you are referring to is found on the search path, you can omit the schema name. However, because table names are not required to be unique within a database, you may find that there are two tables with the same name within your search path (or a table may not be in your search path at all). In those circumstances, you can include the schema name to remove any ambiguity.

To view the schema search path, use the command SHOW SEARCH_PATH:

```
movies=# SHOW SEARCH_PATH;
search_path
--------------
 $user,public
(1 row)
```

The default search path, shown here, is $user,public. The $user part equates to your PostgreSQL user name. For example, if I connect to psql as user bruce, my search path is bruce,public. If a schema named bruce does not exist, PostgreSQL will just ignore that part of the search path and move on to the schema named public. To change the search path, use SET SEARCH_PATH TO:

```
movies=# SET SEARCH_PATH TO 'bruce','sheila','public';
SET
```

You create a new schema with the `CREATE SCHEMA` command and destroy a schema with the `DROP SCHEMA` command:

```
movies=# CREATE SCHEMA bruce;
CREATE SCHEMA

movies=# CREATE TABLE bruces_table( pkey INTEGER );
CREATE TABLE

movies=# \d
      List of relations
 Name           | Schema | Type  | Owner
----------------+--------+-------+-------
 bruces_table   | bruce  | table | bruce
 tapes          | public | table | bruce
(2 rows)

movies=# DROP SCHEMA bruce;
ERROR:  Cannot drop schema bruce because other objects depend on it
        Use DROP ... CASCADE to drop the dependent objects too

movies=# DROP SCHEMA bruce CASCADE;
NOTICE:  Drop cascades to table bruces_table
DROP SCHEMA
```

Notice that you won't be able to drop a schema that is not empty unless you include the `CASCADE` clause. Schemas are a relatively new feature that first appeared in PostgreSQL version 7.3. Schemas are *very* useful. At many sites, you may need to keep a "development" system and a "production" system. You might consider keeping both systems in the same database, but in separate schemas. Another (particularly clever) use of schemas is to separate financial data by year. For example, you might want to keep one year's worth of data per schema. The table names (`invoices`, `sales`, and so on) remain the same across all schemas, but the schema name reflects the year to which the data applies. You could then refer to data for 2001 as FY2001.invoices, FY2001.sales, and so on. The data for 2002 would be stored in FY2002.invoices, FY2002.sales, and so on. This is a difficult problem to solve without schemas because PostgreSQL does not support cross-database access. In other words, if you are connected to database `movies`, you can't access tables stored in another database. Starting with PostgreSQL 7.3, you can keep all your data in a single database and use schemas to partition the data.

When you create a schema, you can specify an optional tablespace—by default, tables created within the schema will be stored in the schema's tablespace. We discuss tablespaces in more detail in the nextwith the `CREATE SCHEMA` section.

Tablespaces

Starting with PostgreSQL version 8.0, you can store database objects (tables and indexes) in alternate locations using a new feature called a *tablespace*. A tablespace is a name that

you give to some directory within your computer's filesystem. Once you create a tablespace (we'll show you how in a moment), you can create schemas, tables, and indexes within that tablespace. A tablespace is defined within a single cluster—all databases within a cluster can refer to the same tablespace.

To create a new tablespace, use the CREATE TABLESPACE command:

```
CREATE TABLESPACE tablespacename
  [ OWNER username ]
  LOCATION 'directory'
```

The *tablespacename* parameter must satisfy the normal rules for all identifiers; it must be 63 characters or shorter and must start with a letter (or the name must be quoted). In addition, you can't create a tablespace whose name begins with the characters 'pg_' since those names are reserved for the PostgreSQL development team. If you omit the OWNER username clause, the new tablespace is owned by the user executing the CREATE TABLESPACE command. By default, you can't create an object in a tablespace unless you are the owner of that tablespace (or you are a PostgreSQL superuser). You can grant CREATE privileges to other users with the GRANT command (see Chapter 23, "Security" for more information on the GRANT command).

The interesting part of a CREATE TABLESPACE command is the *LOCATION 'directory'* clause. The LOCATION clause includes a directory—objects created within the tablespace are stored in that directory. There are a few rules that you must follow before you can create a tablespace:

- You must be a PostgreSQL superuser
- PostgreSQL must be running on a system that supports symbolic links (that means you can't create tablespaces on a Windows host)
- The *directory* must already exist (PostgreSQL won't create the directory for you)
- The *directory* must be empty
- The *directory* name must be shorter than 991 characters
- The *directory* must be owned by the owner of the postmaster process (typically a user named postgres)

If all of those conditions are satisfied, PostgreSQL creates the new tablespace.

When you create a tablespace, the PostgreSQL server performs a number of actions behind the scenes. First, the permissions on the *directory* are changed to 700 (read, write, and execute permissions for the directory owner, all other permissions denied). Next, PostgreSQL creates a single file named PG_VERSION in the given directory (the PG_VERSION file stores the version number of the PostgreSQL server that created the tablespace—if the PostgreSQL developers change the structure of a tablespace in a future version, PG_VERSION will help any conversion tools understand the structure of an existing tablespace). If the permission change succeeds, PostgreSQL adds a new row to the pg_tablespace table (a cluster-wide table) and assigns a new OID (object-id) to that row. Next, the server uses the OID to create a symbolic link between your cluster and the given *directory*. For example, consider the following scenario:

```
movies# CREATE TABLESPACE mytablespace LOCATION '/fastDrive/pg';
CREATE TABLESPACE

movies# SELECT oid, spcname, spclocation
movies-#   FROM
movies-#     pg_tablespace
movies-#   WHERE
movies-#     spcname = 'mytablespace';
  oid  |   spcname    | spclocation
-------+--------------+--------------
 34281 | mytablespace | /fastDrive/pg
```

In this case, PostgreSQL assigned the new tablespace (`mytablespace`) an OID of 34281. PostgreSQL creates a symbolic link that points from `$PGDATA/pg_tblspc/34281` to `/fastDrive/pg`. When you create an object (a table or index) inside of this tablespace, the object is *not* created directly inside of the `/fastDrive/pg` directory. Instead, PostgreSQL creates a subdirectory in the tablespace and then creates the object within that subdirectory. The name of the subdirectory corresponds to the OID of the database (that is, the object-id of the database's entry in the `pg_database` table) that holds the new object. If you create a new table within the `mytablespace` tablespace, like this:

```
movies# CREATE TABLE foo ( data VARCHAR ) TABLESPACE mytablespace;
CREATE TABLE
```

Then find the OID of the new table and the OID of the database (`movies`):

```
movies# SELECT oid FROM pg_class WHERE relname = 'foo';

  oid
-------
 34282
(1 row)

movies# SELECT oid FROM pg_database WHERE datname = 'movies';
  oid
-------
 17228
(1 row)
```

You can see the relationships between the tablespace, the database subdirectory, and the new table:

```
$ ls -l $PGDATA/pg_tblspc
total 0
lrwxrwxrwx    1 postgres postgres       12 Nov  9 19:31 34281 -> /fastDrive/pg

$ ls -l /fastDrive/pg
total 8
drwx------    2 postgres postgres     4096 Nov  9 19:50 17228
-rw-------    1 postgres postgres        4 Nov  9 19:31 PG_VERSION
```

```
$ ls -l /fastDrive/pg/17228
total 0
-rw-------    1 postgres postgres        0 Nov  9 19:50 34282
```

Notice that `$PGDATA/pg_tblspc/34281` is a symbolic link that points to `/fastDrive/pg` (`34281` is the OID of `mytablespace`'s entry in the `pg_tablespace` table), PostgreSQL has created a subdirectory (`17228`) for the movies database, and the table named `foo` was created in that subdirectory (in a file whose name, `34282`, corresponds to the table's OID). By creating a subdirectory for each database, PostgreSQL ensures that you can safely store objects from multiple databases within the same tablespace without worrying about OID collisions.

When you create a cluster (which is done for you automatically when you install PostgreSQL), PostgreSQL silently creates two tablespaces for you: `pg_default` and `pg_global`. PostgreSQL creates objects in the `pg_default` tablespace when it can't find a more appropriate tablespace. The `pg_default` tablespace is always located in the `$PGDATA/base` directory. The `pg_global` tablespace stores cluster-wide tables like `pg_database`, `pg_group`, and `pg_tablespace`—you can't create objects in the `pg_global` tablespace.

The name of the `pg_default` tablespace can be a bit misleading. You may think that PostgreSQL always creates an object in `pg_default` if you omit the `TABLESPACE` *`tablespacename`* clause, but that's not the case. Instead, PostgreSQL follows an inheritance hierarchy to find the appropriate tablespace. If you specify a `TABLESPACE` *`tablespacename`* clause when you execute a `CREATE TABLE` or `CREATE INDEX` command, the server creates the object in the given *`tablespacename`*. If you don't specify a tablespace and you're creating an index, the index is created in the tablespace of the parent table (that is, the table that you are indexing). If you don't specify a tablespace and you're creating a table, the table is created in the tablespace of the parent schema. If you are creating a schema and you don't specify a tablespace, the schema is created in the tablespace of the parent database. If you are creating a database and you don't specify a tablespace, the database is created in the tablespace of the template database (typically, `template1`). So, an index inherits its tablespace from the parent table, a table inherits its tablespace from the parent schema, a schema inherits its tablespace from the parent database, and a database inherits its database from the template database.

To view the databases defined in a cluster, use the `\db` (or `\db+`) command in `psql`:

```
movies=# \db+
                    List of tablespaces
     Name     |   Owner  |     Location     | Access privileges
--------------+----------+------------------+------------------
 mytablespace | postgres | /fastDrive/pg    |
 pg_default   | postgres |                  |
 pg_global    | postgres |                  | {pg=C/pg}
(4 rows)
```

To see a list of objects defined with a given tablespace, use the following query:

```
SELECT relname FROM pg_class
    WHERE reltablespace =
    (
      SELECT oid FROM pg_tablespace WHERE spcname = 'tablespacename'
    );
```

Don't confuse schemas and tablespaces—they both provide organization for the tables and indexes in your cluster, but they are definitely not the same thing. A tablespace affects the *physical* organization of data within a cluster (that is, it a tablespace defines *where* your data is stored). A schema affects the *logical* organization of data within a database—a schema affects name resolution; a tablespace does not. A schema acts as a part of a name; once you've created an object, you can ignore its physical location (its tablespace).

Creating New Databases

Now let's see how to create a new database and how to remove an existing one.

The syntax for the `CREATE DATABASE` command is

```
CREATE DATABASE database-name
    [ WITH [ OWNER    [=] {username|DEFAULT} ]
           [ TEMPLATE [=] {template-name|DEFAULT} ]
           [ ENCODING [=] {encoding|DEFAULT} ]
           [ TABLESPACE [=] tablespace ]]
```

As I mentioned earlier, the `database-name` must follow the PostgreSQL naming rules described earlier and must be unique within the cluster.

If you don't include the `OWNER=username` clause or you specify `OWNER=DEFAULT`, you become the owner of the database. If you are a PostgreSQL superuser, you can create a database that will be owned by another user using the `OWNER=username` clause. If you are not a PostgreSQL superuser, you can still create a database if you have the `CREATEDB` privilege, but you cannot assign ownership to another user. Chapter 21, "PostgreSQL Administration," describes the process of defining user privileges.

The `TEMPLATE=template-name` clause is used to specify a *template* database. A *template* defines a starting point for a database. If you don't include a `TEMPLATE=template-name` or you specify `TEMPLATE=DEFAULT`, the database named `template1` is copied to the new database. All tables, views, data types, functions, and operators defined in the template database are duplicated into the new database. If you add objects (usually functions, operators, and data types) to the `template1` database, those objects will be propagated to any new databases that you create based on `template1`. You can also trim down a template database if you want to reduce the size of new databases. For example, you might decide to remove the geometric data types (and the functions and operators that support that type) if you know that you won't need them. Or, if you have a set of functions that are required by your application, you can define the functions in the `template1` database and all new databases will automatically include those functions. If you want to create an *as-distributed* database, you can use `template0` as your template database. The `template0` database is the starting point for `template1` and contains only the standard objects included in a

PostgreSQL distribution. You should not make changes to the `template0` database, but you can use the `template1` database to provide a site-specific set of default objects.

You can use the `ENCODING=character-set` clause to choose an encoding for the string values in the new database. An *encoding* determines how the bytes that make up a string are interpreted as characters. For example, specifying `ENCODING=SQL_ASCII` tells PostgreSQL that characters are stored in ASCII format, whereas `ENCODING=ISO-8859-8` requests ECMA-121 Latin/Hebrew encoding. When you create a database, all characters stored in that database are encoded in a single format. When a client retrieves data, the client/server protocol automatically converts between the database encoding and the encoding being used by the client. Chapter 22, "Internationalization and Localization," discusses encoding schemes in more detail.

The `TABLESPACE=`*`tablespace-name`* clause tells PostgreSQL that you want to create the database in an alternate location (that is, the database should not be created in the usual `$PGDATA/base` directory). You must create a tablespace before you can use it. If you don't include a `TABLESPACE` clause in the `CREATE DATABASE` command, the new database is created in the same tablespace as the template database.

If you're using an older version of PostgreSQL (older than 8.0), you can't use tablespaces to create a database in a non-standard location. Instead, you must use a feature known as a *location*. In versions of PostgreSQL older than 8.0, the last option for the `CREATE DATABASE` command is the `LOCATION=path` clause. In most cases, you will never have to use the `LOCATION` option, which is good because it's a little strange.

If you do have need to use an alternate location, you will probably want to specify the location by using an environment variable. The environment variable must be known to the `postmaster` processor at the time the `postmaster` is started and it should contain an absolute pathname.

The `LOCATION=path` clause can be confusing. The `path` might be specified in three forms:

- The path contains a `/`, but does not begin with a `/`—this specifies a relative path
- The path begins with a `/`—this specifies an absolute path
- The path does not include a `/`

Relative locations are not allowed by PostgreSQL, so the first form is invalid.

Absolute paths are allowed only if you defined the C/C++ preprocessor symbol "`ALLOW_ABSOLUTE_DBPATHS`" at the time you compiled your copy of PostgreSQL. If you are using a prebuilt version of PostgreSQL, the chances are pretty high that this symbol was *not* defined and therefore absolute paths are not allowed.

So, the only form that you can rely on in a standard distribution is the last—a path that does not include any "`/`" characters. At first glance, this may look like a relative path that is only one level deep, but that's not how PostgreSQL sees it. In the third form, the path must be the *name* of an environment variable. As I mentioned earlier, the environment variable must be known to the `postmaster` processor at the time the `postmaster` is started, and it should contain an absolute pathname. Let's look at an example:

```
$ export PG_ALTERNATE=/bigdrive/pgdata
$ initlocation PG_ALTERNATE
$ pg_ctl restart -l /tmp/pg.log -D $PGDATA
...
$ psql -q -d movies
movies=# CREATE DATABASE bigdb WITH LOCATION=PG_ALTERNATE;
...
```

First, I've defined (and exported) an environment variable named `PG_ALTERNATE`. I've defined `PG_ALTERNATE` to have a value of `/bigdrive/pgdata`—that's where I want my new database to reside. After the environment variable has been defined, I need to initialize the directory structure—the `initlocation` script will take care of that for me. Now I have to restart the `postmaster` so that it can see the `PG_ALTERNATE` variable. Finally, I can start `psql` (or some other client) and execute the `CREATE DATABASE` command specifying the `PG_ALTERNATE` environment variable.

This all sounds a bit convoluted, and it is. The PostgreSQL developers consider it a security risk to allow users to create databases in arbitrary locations. Because the `postmaster` must be started by a PostgreSQL administrator, only an administrator can choose where databases can be created. So, to summarize the process:

1. Create a new environment variable and set it to the path where you want new databases to reside.
2. Initialize the new directory using the `initlocation` application.
3. Stop and restart the `postmaster`.
4. Now, you can use the environment variable with the `LOCATION=path` clause.

createdb

The `CREATE DATABASE` command creates a new database from within a PostgreSQL client application (such as `psql`). You can also create a new database from the operating system command line. The `createdb` command is a shell script that invokes `psql` for you and executes the `CREATE DATABASE` command for you. For more information about `createdb`, see the *PostgreSQL Reference Manual* or invoke `createdb` with the `--help` flag:

```
$ createdb --help
createdb creates a PostgreSQL database.

Usage:
  createdb [OPTION]... [DBNAME] [DESCRIPTION]

Options:
  -D, --tablespace=TABLESPACE  default tablespace for the database
  -E, --encoding=ENCODING      encoding for the database
  -O, --owner=OWNER            database user to own the new database
  -T, --template=TEMPLATE      template database to copy
  -e, --echo                   show the commands being sent to the server
  -q, --quiet                  don't write any messages
```

```
  --help                     show this help, then exit
  --version                  output version information, then exit

Connection options:
  -h, --host=HOSTNAME        database server host or socket directory
  -p, --port=PORT            database server port
  -U, --username=USERNAME    user name to connect as
  -W, --password             prompt for password

By default, a database with the same name as the current user is created.

Report bugs to <pgsql-bugs@postgresql.org>.
```

Dropping a Database

Getting rid of an old database is easy. The `DROP DATABASE` command will delete all of the data in a database and remove the database from the cluster.

For example:

```
movies=# CREATE DATABASE redshirt;
CREATE DATABASE
movies=# DROP DATABASE redshirt;
DROP DATABASE
```

There are no options to the `DROP DATABASE` command; you simply include the name of the database that you want to remove. There *are* a few restrictions. First, you must own the database that you are trying to drop, or you must be a PostgreSQL superuser. Next, you cannot drop a database from within a transaction block—you cannot roll back a `DROP DATABASE` command. Finally, the database must not be in use, even by you. This means that before you can drop a database, you must connect to a different database (`template1` is a good candidate). An alternative to the `DROP DATABASE` command is the `dropdb` shell script. `dropdb` is simply a wrapper around the `DROP DATABASE` command; see the *PostgreSQL Reference Manual* for more information about `dropdb`.

Viewing Databases

Using `psql`, there are two ways to view the list of databases. First, you can ask `psql` to simply display the list of databases and then exit. The `-l` option does this for you:

```
$ psql -l
     List of databases
   Name    |     Owner   | Encoding
-----------+-------------+----------
 template0 | postgres    | UNICODE
 template1 | postgres    | UNICODE
 movies    | bruce       | UNICODE
(3 rows)
$
```

From within psql, you can use the \l or \l+ meta-commands to display the databases within a cluster:

```
movies=# \l+
                    List of databases
   Name    |     Owner     | Encoding |       Description
-----------+---------------+----------+--------------------------
 template0 | postgres      | UNICODE  |
 template1 | postgres      | UNICODE  | Default template database
 movies    | bruce         | UNICODE  | Virtual Video database
 (3 rows)
```

Creating New Tables

The previous section described how to create and drop databases. Now let's move down one level in the PostgreSQL storage hierarchy and talk about creating and dropping tables.

You've created some simple tables in the first two chapters; it's time to talk about some of the more advanced features of the CREATE TABLE command. Here is the command that you used to create the customers table:

```
CREATE TABLE customers (
        customer_id   INTEGER UNIQUE,
        customer_name VARCHAR(50),
        phone         CHAR(8),
        birth_date    DATE,
        balance       DECIMAL(7,2)
);
```

This command creates a *permanent* table named customers. A table name must meet the naming criteria described earlier in this chapter. When you create a table, PostgreSQL automatically creates a new data type[2] with the same name as the table. This means that you can't create a table whose name is the same as an existing data type.

When you execute this command, the customers table is created in the database that you are connected to. If you are using PostgreSQL 7.3 or later, the customers table is created in the first schema in your search path. (If you are using a version older than 7.3, your copy of PostgreSQL does not support schemas). If you want the table to be created in some other schema, you can prefix the table name with the schema qualifier, for example:

```
CREATE TABLE joes_video.customers( ... );
```

The new table is owned by you. You can't give ownership to another user at the time you create the table, but you can change it later using the ALTER TABLE...OWNER TO command (described later).

When you create a table (or an index), you can tell PostgreSQL to store the object in a specific tablespace by including a TABLESPACE *tablespacename* clause, like this:

[2] This seems to be a holdover from earlier days. You can't actually do anything with this data type.

```
CREATE TABLE joes_video.customers( ... ) TABLESPACE mytablespace;
```

If you don't specify a tablespace, PostgreSQL creates the table in the tablespace assigned to the schema (if you're creating an index without specifying a tablespace, the index is created in the tablespace of the parent table).

Temporary Tables

I mentioned earlier that the `customers` table is a permanent table. You can also create *temporary* tables. A permanent table persists after you terminate your PostgreSQL session; a temporary table is automatically destroyed when your PostgreSQL session ends. Temporary tables are also local to your session, meaning that other PostgreSQL sessions can't see temporary tables that you create. Because temporary tables are local to each session, you don't have to worry about colliding with the name of a table created by another session.

If you create a temporary table with the same name as a permanent table, you are effectively *hiding* the permanent table. For example, let's create a temporary table that hides the permanent `customers` table:

```
CREATE TEMPORARY TABLE customers (
        customer_id   INTEGER UNIQUE,
        customer_name VARCHAR(50),
        phone         CHAR(8),
        birth_date    DATE,
        balance       DECIMAL(7,2)
);
```

Notice that the only difference between this command and the command that you used to create the permanent `customers` table is the `TEMPORARY` keyword[3]. Now you have two tables, each named `customers`. If you now `SELECT` from or `INSERT` into the `customers` table, you will be working with the temporary table. Prior to version 7.3, there was no way to get back to the permanent table except by dropping the temporary table:

```
movies=# SELECT * FROM customers;
 customer_id |    customer_name     |  phone   | birth_date | balance
-------------+----------------------+----------+------------+---------
           1 | Jones, Henry         | 555-1212 | 1970-10-10 |    0.00
           2 | Rubin, William       | 555-2211 | 1972-07-10 |   15.00
           3 | Panky, Henry         | 555-1221 | 1968-01-21 |    0.00
           4 | Wonderland, Alice N. | 555-1122 | 1969-03-05 |    3.00
           8 | Wink Wankel          | 555-1000 | 1988-12-25 |    0.00
(5 rows)

movies=# CREATE TEMPORARY TABLE customers
movies-# (
movies(#   customer_id   INTEGER UNIQUE,
movies(#   customer_name VARCHAR(50),
```

[3] You can abbreviate `TEMPORARY` to `TEMP`.

```
movies(#   phone         CHAR(8),
movies(#   birth_date    DATE,
movies(#   balance       DECIMAL(7,2)
movies(#   );
CREATE

movies=# SELECT * FROM customers;
 customer_id |    customer_name     |  phone   | birth_date | balance
-------------+----------------------+----------+------------+---------
(0 rows)

movies=# DROP TABLE customers;
DROP

movies=# SELECT * FROM customers;
 customer_id |    customer_name     |  phone   | birth_date | balance
-------------+----------------------+----------+------------+---------
           1 | Jones, Henry         | 555-1212 | 1970-10-10 |    0.00
           2 | Rubin, William       | 555-2211 | 1972-07-10 |   15.00
           3 | Panky, Henry         | 555-1221 | 1968-01-21 |    0.00
           4 | Wonderland, Alice N. | 555-1122 | 1969-03-05 |    3.00
           8 | Wink Wankel          | 555-1000 | 1988-12-25 |    0.00
(5 rows)
```

Starting with release 7.3, you can access the permanent table by including the name of the schema where the permanent table resides.

A temporary table is like a scratch pad. You can use a temporary table to accumulate intermediate results. Quite often, you will find that a complex query can be formulated more easily by first extracting the data that interests you into a temporary table. If you find that you are creating a given temporary table over and over again, you might want to convert that table into a view. See the section titled "Using Views" in Chapter 1 for more information about views.

Table Constraints

In Chapter 2, "Working with Data in PostgreSQL," we explored the various constraints that you can apply to a column: `NOT NULL`, `UNIQUE`, `PRIMARY KEY`, `REFERENCES`, and `CHECK()`. You can also apply constraints to a table as a whole or to groups of columns within a table.

First, let's look at the `CHECK()` constraint. The syntax for a `CHECK()` constraint is

```
[CONSTRAINT constraint-name] CHECK( boolean-expression )
```

When you define a `CHECK()` constraint for a table, you are telling PostgreSQL that any insertions or updates made to the table must satisfy the `boolean-expression` given within the constraint. The difference between a column constraint and a table constraint is that a column constraint should refer only to the column to which it relates. A table constraint can refer to any column in the table.

For example, suppose that you had an `orders` table to track customer orders:

```
CREATE TABLE orders
(
    customer_number    INTEGER,
    part_number        CHAR(8),
    quantity_ordered   INTEGER,
    price_per_part     DECIMAL(7,2)
);
```

You could create a table-related `CHECK()` constraint to ensure that the extended price (that is, `quantity_ordered` times `price_per_part`) of any given order is at least $5.00:

```
CREATE TABLE orders
(
    customer_number    INTEGER,
    part_number        CHAR(8),
    quantity_ordered   INTEGER,
    price_per_part     DECIMAL(7,2),

    CONSTRAINT verify_minimum_order
     CHECK (( price_per_part * quantity_ordered) >= 5.00::DECIMAL )
);
```

Each time a row is inserted into the orders table (or the `quantity_ordered` or `price_per_part` columns are updated), the `verify_minimum_order` constraint is evaluated. If the expression evaluates to `FALSE`, the modification is rejected. If the expression evaluates to `TRUE` or `NULL`, the modification is allowed.

You may have noticed that a table constraint looks very much like a column constraint. PostgreSQL can tell the difference between the two types by their placement within the `CREATE TABLE` statement. A column constraint is placed *within* a column definition—after the column's data type and before the comma. A table constraint is listed *outside* of a column definition. The only tricky spot is a table constraint that follows the last column definition; you normally would not include a comma after the last column. If you want a constraint to be treated as a table constraint, be sure to include a comma following the last column definition. At the moment, PostgreSQL does not treat table constraints and column constraints differently, but in a future release it may.

Each of the table constraint varieties is related to a type of column constraint.

The `UNIQUE` table constraint is identical to the `UNIQUE` column constraint, except that you can specify that a group of columns must be unique. For example, here is the `rentals` table as currently defined:

```
CREATE TABLE rentals
(
    tape_id     CHARACTER(8),
    customer_id INTEGER,
    rental_date DATE
);
```

Let's modify this table to reflect the business rule that any given tape cannot be rented twice on the same day:

```
CREATE TABLE rentals
(
    tape_id     CHARACTER(8),
    customer_id INTEGER,
    rental_date DATE,

    UNIQUE( rental_date, tape_id )

);
```

Now when you insert a row into the `rentals` table, PostgreSQL will ensure that there are no other rows with the same combination of `rental_date` and `tape_id`. Notice that I did not provide a constraint name in this example; constraint names are optional.

The `PRIMARY KEY` table constraint is identical to the `PRIMARY KEY` column constraint, except that you can specify that the key is composed of a group of columns rather than a single column.

The `REFERENCES` table constraint is similar to the `REFERENCES` column constraint. When you create a `REFERENCES` column constraint, you are telling PostgreSQL that a column value in one table refers to a row in another table. More specifically, a `REFERENCES` column constraint specifies a relationship between two columns. When you create a `REFERENCES` table constraint, you can relate a group of columns in one table to a group of columns in another table. Quite often, you will find that the unique identifier for a table (that is, the `PRIMARY KEY`) is composed of multiple columns. Let's say that the Virtual Video Store is having great success and you decide to open a second store. You might want to consolidate the data for each store into a single database. Start by creating a new table:

```
CREATE TABLE stores
(
    store_id     INTEGER PRIMARY KEY,
    location     VARCHAR
);
```

Now, change the definition of the `customers` table to include a `store_id` for each customer:

```
CREATE TABLE customers (
        store_id      INTEGER REFERENCES stores( store_id ),
        customer_id   INTEGER UNIQUE,
        customer_name VARCHAR(50),
        phone         CHAR(8),
        birth_date    DATE,
        balance       DECIMAL(7,2),

        PRIMARY KEY( store_id, customer_id )
);
```

The store_id column in the customers table refers to the store_id column in the stores table. Because store_id is the primary key to the stores table, you could have written the REFERENCES constraint in either of two ways:

```
store_id INTEGER REFERENCES stores( store_id )
```

or

```
store_id INTEGER REFERENCES stores
```

Also, notice that the primary key for this table is composed of two columns: store_id and customer_id. I can have two customers with the same customer_id as long as they have different store_ids.

Now you have to change the rentals table as well:

```
CREATE TABLE rentals
(
    store_id    INTEGER,
    tape_id     CHARACTER(8),
    customer_id INTEGER,
    rental_date DATE,

    UNIQUE( rental_date, tape_id )
    FOREIGN KEY( store_id, customer_id ) REFERENCES customers
);
```

The customers table has a two-part primary key. Each row in the rentals table refers to a row in the customers table, so the FOREIGN KEY constraint must specify a two-part foreign key. Again, because foreign key refers to the primary key of the customers table, I can write this constraint in either of two forms:

```
FOREIGN KEY( store_id, customer_id )
  REFERENCES customers( store_id, customer_id )
```

or

```
FOREIGN KEY( store_id, customer_id )
 REFERENCES customers
```

Now that I have the referential integrity constraints defined, they will behave as described in the "Column Constraints" section of Chapter 2. Remember, a table constraint functions the same as a column constraint, except that table constraints can refer to more than one column.

Dropping Tables

Dropping a table is much easier than creating a table. The syntax for the DROP TABLE command is

```
DROP TABLE table-name [, ...];
```

If you are using PostgreSQL 7.3 or later, you can qualify the table name with a schema. For example, here is the command to destroy the `rentals` table:

```
DROP TABLE rentals;
```

If the `rentals` table existed in some schema other than your current schema, you would qualify the table name:

```
DROP TABLE sheila.rentals;
```

You can destroy a table only if you are the table's owner or if you are a PostgreSQL superuser. Notice that I used the word *destroy* here rather than *drop*. It's important to realize that when you execute a `DROP TABLE` command, you are destroying all the data in that table.

PostgreSQL has a nice feature that I have not seen in other databases: You can roll back a `DROP TABLE` command. Try the following experiment. First, let's view the contents of the `tapes` table:

```
movies=# SELECT * FROM tapes;

tape_id  |     title     | duration
----------+---------------+----------
 AB-12345 | The Godfather |
 AB-67472 | The Godfather |
 MC-68873 | Casablanca    |
 OW-41221 | Citizen Kane  |
 AH-54706 | Rear Window   |
(5 rows)
```

Now, start a multistatement transaction and destroy the `tapes` table:

```
movies=# BEGIN WORK;
BEGIN

movies=# DROP TABLE tapes;

NOTICE: DROP TABLE implicitly drops referential integrity trigger
         from table "rentals"
DROP
```

If you try to `SELECT` from the `tapes` table, you'll find that it has been destroyed:

```
movies=# SELECT * FROM tapes;
ERROR:  Relation "tapes" does not exist
```

If you `COMMIT` this transaction, the table will permanently disappear; let's `ROLLBACK` the transaction instead:

```
movies=# ROLLBACK;
ROLLBACK
```

The `ROLLBACK` threw out all changes made since the beginning of the transaction, including the `DROP TABLE` command. You should be able to `SELECT` from the `tapes` table again and see the same data that was there before:

```
movies=# SELECT * FROM tapes;
tape_id  |     title     | duration
----------+---------------+----------
 AB-12345 | The Godfather |
 AB-67472 | The Godfather |
 MC-68873 | Casablanca    |
 OW-41221 | Citizen Kane  |
 AH-54706 | Rear Window   |
(5 rows)
```

This is a *very* nice feature. You can roll back `CREATE TABLE`, `DROP TABLE`, `CREATE VIEW`, `DROP VIEW`, `CREATE INDEX`, `DROP INDEX`, and so on. I'll discuss transactions a bit later in this chapter. For now, I'd like to point out a few details that I glossed over in the previous example. You may have noticed that the `DROP TABLE` command produced a `NOTICE`.

```
movies=# DROP TABLE tapes;
NOTICE: DROP TABLE implicitly drops referential integrity trigger
        from table "rentals"
DROP
```

When you drop a table, PostgreSQL will automatically `DROP` any indexes defined for that table as well as any triggers or rules. If other tables refer to the table that you dropped (by means of a `REFERENCE` constraint), PostgreSQL will automatically drop the constraints in the other tables. However, any *views* that refer to the dropped table will not be removed—a view can refer to many tables and PostgreSQL would not know how to remove a single table from a multitable `SELECT`.

Inheritance

Another PostgreSQL feature that is uncommon in relational database systems is *inheritance*. Inheritance is one of the foundations of the object-oriented programming paradigm. Using inheritance, you can define a hierarchy of related data types (in PostgreSQL, you define a hierarchy of related tables). Each layer in the inheritance hierarchy represents a *specialization* of the layer above it[4].

Let's look at an example. The Virtual Video database defines a table that stores information about the tapes that you have in stock:

```
movies=# \d tapes
  Column  |         Type          | Modifiers
----------+-----------------------+-----------
 tape_id  | character(8)          | not null
 title    | character varying(80) | not null
 duration | interval              |
```

[4] We'll view an inheritance hierarchy with the most general types at the top and the most specialized types at the bottom.

For each tape, you store the `tape_id`, `title`, and `duration`. Let's say that you decide to jump into the twenty-first century and rent DVDs as well as videotapes. You *could* store DVD records in the `tapes` table, but a tape and a DVD are not really the same thing. Let's create a new table that defines the characteristics common to both DVDs and videotapes:

```
CREATE TABLE video
(
  video_id        CHARACTER(8) PRIMARY KEY,
  title           VARCHAR(80),
  duration        INTERVAL
);
```

Now, create a table to hold the DVDs. For each DVD you have in stock, you want to store everything in the `video` table plus a `region_id` and an array of `audio_tracks`. Here is the new table definition:

```
movies=# CREATE TABLE dvds
movies-# (
movies(#   region_id    INTEGER,
movies(#   audio_tracks VARCHAR[]
movies(# ) INHERITS ( video );
```

Notice the last line in this command: You are telling PostgreSQL that the `dvds` table *inherits from* the `video` table. Now let's `INSERT` a new DVD:

```
movies=# INSERT INTO dvds VALUES
movies=# (
movies(#   'ASIN-750',                      -- video_id
movies(#   'Star Wars',                     -- title
movies(#   '121 minutes',                   -- duration
movies(#   1,                               -- region_id
movies(#   '{English,Spanish}'              -- audio_tracks
movies(# );
```

Now, if you `SELECT` from the `dvds` table, you'll see the information that you just inserted:

```
 video_id |   title   | duration | region_id |   audio_tracks
----------+-----------+----------+-----------+------------------
 ASIN-750 | Star Wars | 02:01:00 |         1 | {English,Spanish}
```

At this point, you might be thinking that the `INHERITS` clause did nothing more than create a row template that PostgreSQL copied when you created the `dvds` table. That's not the case—if you simply want to create a table that has the same structure as another table, use the `LIKE` *`table-name`* clause instead of the `INHERITS` *`table-name`* clause. When we say that `dvds` inherits from `video`, we are not simply saying that a DVD is *like* a video, we are saying that a DVD *is* a video. Let's `SELECT` from the `video` table now; remember, you haven't explicitly inserted any data into the `video` table, so you might expect the result set to be empty:

```
movies=# SELECT * FROM video;
 video_id |   title   | duration
----------+-----------+----------
 ASIN-750 | Star Wars | 02:01:00
```

A DVD is a video. When you `SELECT` from the `video` table, you see only the columns that comprise a `video`. When you `SELECT` from the `dvds` table, you see all the columns that comprise a DVD. In this relationship, you say that the `dvds` table *specializes*[5] the more general `video` table.

If you are using a version of PostgreSQL older than 7.2, you must code this query as `SELECT * FROM video*` to see the DVD entries. Starting with release 7.2, `SELECT` will include descendent tables and you have to say `SELECT * FROM ONLY video` to suppress descendents.

You now have a new table to track your DVD inventory; let's go back and redefine the `tapes` table to fit into the inheritance hierarchy. For each tape, we want to store a `video_id`, a `title`, and a `duration`. This is where we started: the `video` table already stores all this information. You should still create a new table to track videotapes—at some point in the future, you may find information that relates to a videotape, but not to a DVD:

```
movies=# CREATE TABLE tapes ( ) INHERITS( video );
CREATE
```

This `CREATE TABLE` command creates a new table identical in structure to the `video` table. Each row in the `tapes` table will contain a `video_id`, a `title`, and a `duration`. Insert a row into the `tapes` table:

```
movies=# INSERT INTO tapes VALUES
movies-# (
movies(#   'ASIN-8YD',
movies(#   'Flight To Mars(1951)',
movies(#   '72 min'
movies(# );
INSERT
```

When you `SELECT` from the `tapes` table, you should see this new row:

```
movies=# SELECT * FROM tapes;
 tape_id  |        title         | duration
----------+----------------------+----------
 ASIN-8YD | Flight To Mars(1951) | 01:12:00
(1 row)
```

And because a tape *is* a video, you would also expect to see this row in the video table:

```
movies=# SELECT * FROM video;
```

[5] Object-oriented terminology defines many different phrases for this inheritance relationship: specialize/generalize, subclass/superclass, and so on. Choose the phrase that you like.

```
 video_id |             title              | duration
----------+--------------------------------+----------
 ASIN-750 | Star Wars                      | 02:01:00
 ASIN-8YD | Flight To Mars(1951)           | 01:12:00
(2 rows)
```

Now here's the interesting part. A DVD is a video—any row that you add to the `dvds` table shows up in the `video` table. A tape is a video—any row that you add to the `tapes` table shows up in the `video` table. But a DVD is *not* a tape (and a tape is *not* a DVD). Any row that you add to the `dvds` table will *not* show up in the `tapes` table (and vice versa).

If you want a list of all the tapes you have in stock, you can `SELECT` from the `tapes` table. If you want a list of all the DVDs in stock, `SELECT` from the `dvds` table. If you want a list of all videos in stock, `SELECT` from the `videos` table.

In this example, the inheritance hierarchy is only two levels deep. PostgreSQL imposes no limit to the number of levels that you can define in an inheritance hierarchy. You can also create a table that inherits from *multiple* tables—the new table will have all the columns defined in the more general tables.

I should caution you about two problems with the current implementation of inheritance in PostgreSQL. First, indexes are not shared between parent and child tables. On one hand, that's good because it gives you good performance. On the other hand, that's bad because PostgreSQL uses an index to guarantee uniqueness. That means that you could have a videotape and a DVD with the same `video_id`. Of course, you can work around this problem by encoding the type of video in the `video_id` (for example, use a `T` for tapes and a `D` for DVDs). But PostgreSQL won't give you any help in fixing this problem. The other *potential* problem with inheritance is that triggers are not shared between parent and child tables. If you define a trigger for the topmost table in your inheritance hierarchy, you will have to remember to define the same trigger for each descendant.

We have redefined some of the example tables many times in the past two chapters. In a real-world environment, you probably won't want to throw out all your data each time you need to make a change to the definition of an existing table. Let's explore a better way to alter a table.

ALTER TABLE

Now that you have a `video` table, a `dvds` table, and a `tapes` table, let's add a new column to all three tables that you can use to record the rating of the video (PG, G, R, and so on).

You could add the `rating` column to the `tapes` table and to the `dvds` table, but you really want the `rating` column to be a part of every video. The `ALTER TABLE ... ADD COLUMN` command adds a new column for you, leaving all the original data in place:

```
movies=# ALTER TABLE video ADD COLUMN rating VARCHAR;
ALTER
```

Now, if you look at the definition of the `video` table, you will see the new column:

```
movies=# \d video
                Table "video"
  Column  |         Type          | Modifiers
----------+-----------------------+-----------
 video_id | character(8)          | not null
 title    | character varying(80) |
 duration | interval              |
 rating   | character varying     |
Indexes:
    "video_pkey" PRIMARY KEY, btree (video_id)
```

After the `ALTER TABLE` command completes, each row in the `video` table has a new column; the value of every `rating` column will be `NULL`. Because you have changed the definition of a `video`, and a DVD is a `video`, you might expect that the `dvds` table will also contain a `rating` column:

```
movies=# \d dvds
                     Table "dvds"
    Column    |         Type          | Modifiers
--------------+-----------------------+-----------
 video_id     | character(8)          | not null
 title        | character varying(80) |
 duration     | interval              |
 region_id    | integer               |
 audio_tracks | character varying[]   |
 rating       | character varying     |
Inherits: video
```

Similarly, the `tapes` table will also inherit the new `rating` column:

```
movies=# \d tapes
                   Table "tapes"
  Column  |         Type          | Modifiers
----------+-----------------------+-----------
 video_id | character(8)          | not null
 title    | character varying(80) |
 duration | interval              |
 rating   | character varying     |
Inherits: video
```

Starting with PostgreSQL version 8.0, you can change the data type of an existing column using `ALTER TABLE`. For example, to change the data type of the `customers.customer_id` column from `INTEGER` to `NUMERIC( 7, 2 )`, you could execute the command:

```
ALTER TABLE customers ALTER COLUMN customer_id TYPE NUMERIC( 7,2 )
```

As long as PostgreSQL knows how to convert a value from the old data type to the new data type, you can freely change data types. If PostgreSQL doesn't know how to convert between the old and new types, you can include a `USING expression` clause to tell

PostgreSQL how to perform the conversion. The *expression* following the USING keyword typically refers to the original column value. For example, if you want to change the data type of customers.customer_id and multiply each customer_id by 100 at the same time, use the following command:

```
ALTER TABLE customers ALTER COLUMN customer_id TYPE NUMERIC( 7,2 ) USING cus-
tomer_id * 100
```

You can also refer to *other* columns in the USING *expression*. For example, say that you are currently storing each customer name in two columns, last_name and first_name, and you've decided to combine them into a single column named customer name. You can do that with the following commands:

```
movies=# ALTER TABLE customers
movies-#     ALTER COLUMN last_name
movies-#         TYPE VARCHAR USING ( last_name || ',' || first_name ),
movies-#     DROP COLUMN first_name;
ALTER TABLE

movies=# ALTER TABLE customers
movies-#     RENAME COLUMN last_name TO customer_name;
ALTER TABLE
```

The first ALTER TABLE command performs two alterations. First, for each row in the table, it evaluates the expression last_name || ',' || first_name and assigns that value to the last_name column (converting the result into type VARCHAR along the way). Next, the (first) ALTER TABLE command removes the first_name column from each row. You're left with a single column called last_name that contains the concatenation of the original last_name and first_name columns (with a comma in between). The second ALTER TABLE command renames the last_name column to customer_name.

Keep in mind that some ALTER TABLE commands will take longer to execute than others. It takes very little time to change the name of a column. It can take quite a while to change the data type of a column (because PostgreSQL has to traverse every row in the table and write out a new version). If you use ALTER TABLE ... SET TABLESPACE to move a table from one tablespace to another, the server must physically copy each block in the table. In most cases, it's faster to execute a series of ALTER TABLE commands than it is to read the old data into a client application, change each row, and then write the result back to the server. When you use an ALTER TABLE command, the entire transformation occurs within the server; if you modify the structure of a table using a custom-written client application, you have to send every row to the client, perform the transformation, and then send every row back to the server.

The ALTER TABLE command is useful when you are in the development stages of a project. Using ALTER TABLE, you can add new columns to a table, define default values, rename columns (and tables), add and drop constraints, change the data type of a column, and transfer ownership. The capabilities of the ALTER TABLE command seem to grow with each new release—see the *PostgreSQL Reference Manual* for more details.

Adding Indexes to a Table

Most of the tables that you have created so far have no indexes. An index serves two purposes. First, an index can be used to guarantee uniqueness. Second, an index provides quick access to data (in certain circumstances).

Here is the definition of the `customers` table that you created in Chapter 1:

```
CREATE TABLE customers (
        customer_id   INTEGER UNIQUE,
        customer_name VARCHAR(50),
        phone         CHAR(8),
        birth_date    DATE,
        balance       DECIMAL(7,2)
);
```

When you create this table, PostgreSQL will display a rather terse message:

```
NOTICE:  CREATE TABLE / UNIQUE will create implicit
    index 'customers_customer_id_key' for table 'customers'
```

What PostgreSQL is trying to tell you here is that even though you didn't explicitly ask for one, an index has been created on your behalf. The implicit index is created so that PostgreSQL has a quick way to ensure that the values that you enter into the `customer_id` column are unique.

Think about how you might design an algorithm to check for duplicate values in the following list of names:

Grumby, Jonas

Hinkley, Roy

Wentworth, Eunice

Floyd, Heywood

Bowman, David

Dutton, Charles

Poole, Frank

Morbius, Edward

Farman, Jerry

Stone, Jeremy

Dutton, Charles

Manchek, Arthur

A first attempt might simply start with the first value and look for a duplicate later in the list, comparing `Grumby, Jonas` to `Hinkley, Roy`, then `Wentworth, Eunice`, and so on. Next, you would move to the second name in the list and compare `Hinkley, Roy` to `Wentworth, Eunice`, then `Floyd, Heywood`, and so on. This algorithm would certainly work, but it would turn out to be slow as the list grew longer. Each time you add a new name to the list, you have to compare it to every other name already in the list.

A better solution would be to first sort the list:

Bowman, David
Dutton, Charles
Dutton, Charles
Farman, Jerry
Floyd, Heywood
Grumby, Jonas
Hinkley, Roy
Manchek, Arthur
Morbius, Edward
Poole, Frank
Stone, Jeremy
Wentworth, Eunice

After the list is sorted, it's easy to check for duplicates—any duplicate values appear next to each other. To check the sorted list, you start with the first name, `Bowman, David` and compare it to the second name, `Dutton, Charles`. If the second name is not a duplicate of the first, you know that you won't find any duplicates later in the list. Now when you move to the second name on the list, you compare it to the third name—now you can see that there is a duplicate. Duplicate values appear next to each other after the list is sorted. Now when you add a new name to the list, you can stop searching for duplicate values as soon as you encounter a value that sorts after the name you are adding.

An index is similar in concept to a sorted list, but it's even better. An index provides a quick way for PostgreSQL to find data within a range of values. Let's see how an index can help narrow a search. First, let's assign a number to each of the names in the sorted list, just for easy reference (I've removed the duplicate value):

1 Bowman, David
2 Dutton, Charles
3 Farman, Jerry
4 Floyd, Heywood
5 Grumby, Jonas
6 Hinkley, Roy
7 Manchek, Arthur
8 Morbius, Edward
9 Poole, Frank
10 Stone, Jeremy
11 Wentworth, Eunice

Now let's build a (simplistic) index (see Figure 3.2). The English alphabet contains 26 letters—split this roughly in half and choose to keep track of where the "Ms" start in the list. In this list, names beginning with an `M` start at entry number 7. Keep track of this pair (M,7) and call it the *root* of your index.

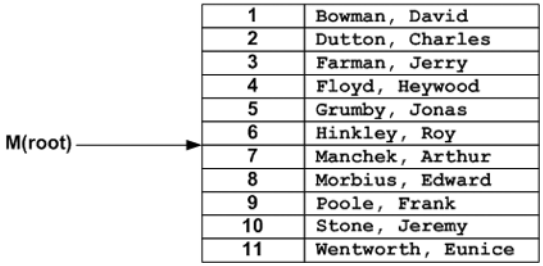

Figure 3.2 One-level index.

Now when you insert a new name, `Tyrell, Eldon`, you start by comparing it to the root. The root of the index tells you that names starting with the letter `M` are found starting at entry number 7. Because the list is sorted, and you know that `Tyrell` will sort after `M`, you can start searching for the insertion point at entry 7, skipping entries 1 through 6. Also, you can stop searching as soon as you encounter a name that sorts later than `Tyrell`.

As your list of names grows, it would be advantageous to add more levels to the index (see Figure 3.3). The letter `M` splits the alphabet (roughly) in half. Add a second level to the index by splitting the range between `A` and `M` (giving you `G`), and splitting the range between `M` and `Z` (giving you `T`).

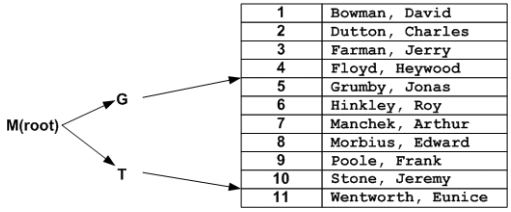

Figure 3.3 Two-level index.

Now when you want to add `Tyrell, Eldon` to the list, you compare `Tyrell` against the root and find that `Tyrell` sorts later than `M`. Moving to the next layer of the index, you find that `Tyrell` sorts later than `T`, so you can jump straight to slot number 11 and insert the new value.

You can see that you can add as many index levels as you need. Each level divides the parent's range in half, and each level reduces the number of names that you have to search to find an insertion point[6].

Using an index is similar in concept to the way you look up words in a dictionary. If you have a dictionary handy, pull it off the shelf and take a close look at it. If it's like my dictionary, it has those little thumb-tab indentations, one for each letter of the alphabet. If I want to find the definition of the word "polyglot," I'll find the thumb-tab labeled "*P*" and start searching about halfway through that section. I know, because the dictionary is sorted, that "polyglot" won't appear in any section prior to "P" and it won't appear in any section following "P." That little thumb-tab saves a lot of searching.

You also can use an index as a quick way to check for uniqueness. If you are inserting a new name into the index structure shown earlier, you simply search for the new name in the index. If you find it in the index, it is obviously a duplicate.

I mentioned earlier that PostgreSQL uses an index for two purposes. You've seen that an index can be used to search for unique values. But how does PostgreSQL use an index to provide faster data access?

Let's look at a simple query:

```
SELECT * FROM characters WHERE name >= 'Grumby' AND name < 'Moon';
```

Now assume that the list of names that you worked with before is actually a table named `characters` and you have an index defined for the `name` column, as in Figure 3.4.

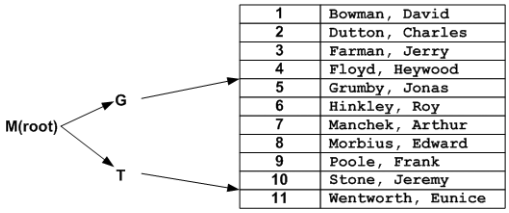

Figure 3.4 Two-level index (again).

When PostgreSQL parses through the `SELECT` statement, it notices that you are constraining the result set to a *range* of names and that you have an index on the `name` column. That's a convenient combination. To satisfy this statement, PostgreSQL can use the index to start searching at entry number 5. Because the rows are already sorted, PostgreSQL can stop searching as soon as it finds the first entry greater than "Moon" (that is, the search ends as soon as you hit entry number 8). This kind of operation is called a *partial index scan*.

[6] Technically speaking, the index diagrams discussed here depict a clustered index. In a clustered index, the leaf nodes in the index tree are the data rows themselves. In a non-clustered index, the leaf nodes are actually row pointers—the rows are not kept in sorted order. PostgreSQL does not support clustered indexes. I've diagrammed the index trees in clustered form for clarity. A clustered index provides fast, sequential access along one index path, but it is very expensive to maintain.

Think of how PostgreSQL would process this query if the rows were *not* indexed. It would have to start at the beginning of the table and compare each row against the constraints; PostgreSQL can't terminate the search without processing every row in the table. This kind of operation is called a *full table scan*, or *table scan*.

Because this kind of index can access data in sorted order, PostgreSQL can use such an index to avoid a sort that would otherwise be required to satisfy an `ORDER BY` clause.

In these examples, we are working with small tables, so the performance difference between a full table scan and an indexed range read is negligible. As tables become larger, the performance difference can be huge. Chapter 4, "Performance," discusses how the PostgreSQL query optimizer chooses when it is appropriate to use an index.

PostgreSQL actually supports several kinds of indexes. The previous examples show how a B-Tree index works[7]. Another type of index is the Hash index. A Hash index uses a technique called *hashing* to evenly distribute keys among a number of *hash buckets*. Each key value added to a hash index is run through a hashing function. The result of a hashing function is a bucket number. A simplistic hashing function for string values might sum the ASCII value of each character in the string and then compute the sum modulo the number of buckets to get the result. In C, you might write this function as

```
int hash_string( char * key, int bucket_count )
{
    int hash = 0;
    int i;

    for( i = 0; i < strlen( key ); i++ )
        hash = hash + key[i];

    return( hash % bucket_count );
}
```

Let's run each of the names in the `characters` table through this function to see what kind of numbers you get back (I've used a `bucket_count` of 5):

hash_string() Value	Name
1	Grumby, Jonas
2	Hinkley, Roy
3	Wentworth, Eunice
4	Floyd, Heywood
4	Bowman, David
3	Dutton, Charles
3	Poole, Frank

[7] The "B" in B-Tree stands for "Balanced." A balanced tree is a type of data structure that retains its performance characteristics even in the face of numerous insertions and deletions. The most important feature of a B-Tree is that it takes about the same amount of time to find any given record.

hash_string() Value	Name
0	Morbius, Edward
0	Farman, Jerry
0	Stone, Jeremy
4	Manchek, Arthur

The numbers returned don't really have any intrinsic meaning, they simply serve to distribute a set of keys amongst a set of buckets.

Now let's reformat this table so that the contents are grouped by bucket number:

Bucket Number	Bucket Contents
0	Morbius, Edward
	Farman, Jerry
	Stone, Jeremy
1	Grumby, Jonas
2	Hinkley, Roy
3	Wentworth, Eunice

Bucket Number	Bucket Contents
	Dutton, Charles
	Poole, Frank
4	Floyd, Heywood
	Bowman, David
	Manchek, Arthur

You can see that the hash function (`hash_string()`) did a respectable job of distributing the names between the five hash buckets. Notice that we did not have to assign a unique hash value to each key—hash keys are seldom unique. The important feature of a good hash function is that it distributes a set of keys fairly evenly. Now that you have a Hash index, how can you use it? First, let's try to insert a new name: `Lowell, Freeman`. The first thing you do is run this name through your `hash_string()` function, giving you a hash value of 4. Now you know that if `Lowell, Freeman` is already in the index, it will be in bucket number 4; all you have to do is search that one bucket for the name you are trying to insert.

There are a couple of important points to note about Hash indexes.

First, you may have noticed that each bucket can hold many keys. Another way to say this is that each key does not have a unique hash value. If you have too many collisions (that is, too many keys hashing to the same bucket), performance will suffer. A good hash function distributes keys evenly between all hash buckets.

Second, notice that a hash table is not sorted. The name `Floyd, Heywood` hashes to bucket 4, but `Farman, Jerry` hashes to bucket 0. Consider the `SELECT` statement that we looked at earlier:

```
SELECT * FROM characters WHERE name >= 'Grumby' AND name < 'Moon';
```

To satisfy this query using a Hash index, you have to read the entire contents of each bucket. Bucket 0 contains one row that meets the constraints (`Farman, Jerry`), bucket 2 contains one row, and bucket 4 contains one row. A Hash index offers no advantage to a range read. A Hash index is good for searches based on equality. For example, the `SELECT` statement

```
SELECT * FROM characters WHERE name = 'Grumby, Jonas';
```

can be satisfied simply by hashing the string that you are searching for. A Hash index is also useful when you are joining two tables where the join constraint is of the form `table1-column = table2-column`[8]. A Hash read cannot be used to avoid a sort required to satisfy an `ORDER BY` clause.

PostgreSQL supports two other types of index structures: the R-Tree index and the GiST index. An R-Tree index is best suited for indexing spatial (that is, geometric or geographic) data. A GiST index is a B-Tree index that can be extended by defining new query predicates[9]. More information about GiST indexes can be found at http://gist.cs.berkeley.edu/.

Tradeoffs

The previous section showed that PostgreSQL can use an index to speed the process of searching for data within a range of values (or data with an exact value). Most queries (that is, `SELECT` commands) in PostgreSQL include a `WHERE` clause to limit the result set. If you find that you are often searching for results based on a range of values for a specific column or group of columns, you might want to consider creating an index that covers those columns.

However, you should be aware that an index represents a performance tradeoff. When you create an index, you are trading read performance for write performance. An index can significantly reduce the amount of time it takes to retrieve data, but it will also *increase* the amount of time it takes to `INSERT`, `DELETE`, and `UPDATE` data. Maintaining an index introduces substantial overhead when you modify the data within a table.

You should consider this tradeoff when you feel the need to add a new index to a table. Adding an index to a table that is updated frequently will certainly slow the updates. A good candidate for an index is a table that you `SELECT` from frequently but seldom update. A customer list, for example, doesn't change often (possibly several times each day), but you probably query the customer list frequently. If you find that you often query the customer list by phone number, it would be beneficial to index the phone number column. On the other hand, a table that is updated frequently, but seldom queried, such as a transaction history table, would be a poor choice for an index.

[8] This type of join is known as an equi-join.

[9] A predicate is a test. A simple predicate is the less-than operator (<). An expression such as a < 5 tests whether the value of a is less than 5. In this expression, < is the predicate and it is called the less-than predicate. Other predicates are =, >, >=, and so on.

Creating an Index

Now that you have seen what an index can do, let's look at the process of adding an index to a table. The process of creating a new index can range from simple to somewhat complex.

Let's add an index to the `rentals` table. Here is the structure of the rentals table for reference:

```
CREATE TABLE rentals
(
        tape_id      CHARACTER(8) REFERENCES tapes,
        customer_id  INTEGER REFERENCES customers,
        rental_date  DATE
);
```

The syntax for a simple `CREATE INDEX` command is

```
CREATE [UNIQUE] INDEX index-name ON table-name( column [,...] );
```

You want to index the `rental_date` column in the `rentals` table:

```
CREATE INDEX rentals_rental_date ON rentals ( rental_date );
```

You haven't specified any optional information in this command (I'll get to the options in a moment), so PostgreSQL creates a B-Tree index named `rentals_rental_date`. PostgreSQL considers using this whenever it finds a `WHERE` clause that refers to the `rental_date` column using the `<`, `<=`, `=`, `>=`, or `>` operator. This index also can be used when you specify an `ORDER BY` clause that sorts on the `rental_date` column.

Multicolumn Indexes

A B-Tree index (or a GiST index) can cover more than one column. Multicolumn indexes are usually created when you have many values on the second column for each value in the first column. For example, you might want to create an index that covers the `rental_date` and `tape_id` columns—you have many different tapes rented on any given date. PostgreSQL can use multicolumn indexes for selection or for ordering. When you create a multicolumn index, the order in which you name the columns is important. PostgreSQL can use a multicolumn index when you are selecting (or ordering by) a prefix of the key. In this context, a prefix may be the entire key or a leading portion of the key. For example, the command `SELECT * FROM rentals ORDER BY rental_date` could not use an index that covers `tape_id` plus `rental_date`, but it could use an index that covers `rental_date` plus `tape_id`.

The `index-name` must be unique within the database: You can't have two indexes with the same name, even if they are defined on different tables. New rows are indexed as they are added, and deleted rows are removed. If you change the `rental_date` for a given row, the index will be updated automatically. If you have any data in the `rentals` table, each row will be included in the index.

Indexes and `NULL` Values

Earlier, I mentioned that an index includes a pointer for every row in a table. That statement isn't 100% accurate. PostgreSQL will not index `NULL` values in R-Tree, Hash, and GiST indexes. Because such an index

> will never include NULL values, it cannot be used to satisfy the ORDER BY clause of a query that returns all rows in a table. For example, if you define a GiST index covering the phone column in the customers table, that index would not include rows where phone was NULL. If you executed the command SELECT * FROM customers ORDER BY phone, PostgreSQL would have to perform a full table scan and then sort the results. If PostgreSQL tried to use the phone index, it would not find all rows. If the phone column were defined as NOT NULL, then PostgreSQL could use the index to avoid a sort. Or, if the SELECT command included the clause WHERE phone IS NOT NULL, PostgreSQL could use the index to satisfy the ORDER BY clause. An R-Tree, Hash, or GiST index that covers an optional (that is, NULLs-allowed) column will not be used to speed table joins, either.
>
> A B-Tree index (the default index type) *does* include NULL values.

If you don't specify an index type when creating an index, you'll get a B-Tree index. Let's change the rentals_rental_date index into a Hash index. First, drop the original index:

```
DROP INDEX rentals_rental_date;
```

Then you can create a new index:

```
CREATE INDEX rentals_rental_date ON rentals USING HASH ( rental_date );
```

The only difference between this CREATE INDEX command and the previous one is that I have included a USING clause. You can specify USING BTREE (which is the default), USING HASH, USING RTREE, or USING GIST.

This index cannot be used to satisfy an ORDER BY clause. In fact, this index can be used only when rental_date is compared using the = operator.

I dropped the B-Tree index before creating the Hash index, but that is not strictly necessary. It is perfectly valid (but unusual) to have two or more indexes that cover the same column, as long as the indexes are uniquely named. If we had both a B-Tree index and a Hash index covering the rental_date column, PostgreSQL could use the Hash index for = comparisons and the B-Tree index for other comparisons.

Functional Indexes and Partial Indexes

Now let's look at two variations on the basic index types: functional indexes and partial indexes.

A column-based index catalogs the values found in a column (or a set of columns). A functional index (or more precisely a function-valued index) catalogs the values returned by a given function. This might be easiest to understand by looking at an example. Each row in the customers table contains a phone number. You can use the exchange[10] portion of the phone number to determine whether a given customer is located close to your store. For example, you may know that the 555, 556, and 794 exchanges are within five miles of your virtual video store. Let's create a function that extracts the exchange from a phone number:

[10] In the U.S., a phone number is composed of an optional three-digit area code, a three-digit exchange, and a four-digit?ummm, number.

```
--  exchange_index.sql
--
CREATE OR REPLACE FUNCTION get_exchange( CHARACTER )
  RETURNS CHARACTER AS '

  DECLARE
    result               CHARACTER(3);
  BEGIN

    result := SUBSTR( $1, 1, 3 );

    return( result );
  END;
' LANGUAGE 'plpgsql' WITH ( ISCACHABLE );
```

Don't be too concerned if this looks a bit confusing; I'll cover the PL/pgSQL language in more detail in Chapter 7, "PL/pgSQL." This function (`get_exchange()`) accepts a single argument, presumably a phone number, and extracts the first three characters. You can call this function directly from `psql`:

```
movies=# SELECT customer_name, phone, get_exchange( phone )
movies-#   FROM customers;

    customer_name     |  phone   | get_exchange
----------------------+----------+------------
 Jones, Henry         | 555-1212 | 555
 Rubin, William       | 555-2211 | 555
 Panky, Henry         | 555-1221 | 555
 Wonderland, Alice N. | 555-1122 | 555
 Wink Wankel          | 555-1000 | 555
```

You can see that given a phone number, `get_exchange()` returns the first three digits. Now let's create a function-valued index that uses this function:

```
CREATE INDEX customer_exchange ON customers ( get_exchange( phone ));
```

When you insert a new row into a column-based index, PostgreSQL will index the values in the columns covered by that index. When you insert a new row into a *function-valued* index, PostgreSQL will call the function that you specified and then index the return value.

After the `customer_exchange` index exists, PostgreSQL can use it to speed up queries such as

```
SELECT * FROM customers WHERE get_exchange( phone ) = '555';
SELECT * FROM customers ORDER BY get_exchange( phone );
```

Now you have an index that you can use to search the customer list for all customers that are geographically close. Let's pretend that you occasionally want to send advertising flyers to those customers closest to you: you might never use the `customer_exchange`

index for any other purpose. If you need the customer_exchange index for only a small set of customers, why bother maintaining that index for customers outside of your vicinity? This is where a *partial* index comes in handy. When you create an index, you can include a WHERE clause in the CREATE INDEX command. Each time you insert (or update) a row, the WHERE clause is evaluated. If a row satisfies the constraints of the WHERE clause, that row is included in the index; otherwise, the row is not included in the index. Let's DROP the customer_exchange index and replace it with a partial, function-valued index:

```
movies=# DROP INDEX customer_exchange;
DROP
movies=# CREATE INDEX customer_exchange
movies-#   ON customers ( get_exchange( phone ))
movies-#   WHERE
movies-#     get_exchange( phone ) = '555'
movies-#      OR
movies-#     get_exchange( phone ) = '556'
movies-#      OR
movies-#     get_exchange( phone ) = '794';
CREATE
```

Now the customer_exchange partial index contains entries only for customers in the 555, 556, or 794 exchange.

There are three performance advantages to a partial index:

- A partial index requires less disk space than a full index.
- Because fewer rows are cataloged in a partial index, the cost of maintaining the index is lower.
- When a partial index is used in a query, PostgreSQL will have fewer index entries to search.

Partial indexes and function-valued indexes are variations on the four basic index types. You can create a function-valued Hash index, B-Tree index, R-tree index, or GiST index. You can also create a partial variant of any index type. And, as you have seen, you can create partial function-valued indexes (of any type). A function-valued index doesn't change the organization of an index—just the values that are actually included in the index. The same is true for a partial index.

Creating Indexes on Array Values

Most indexes cover scalar-valued columns (columns that store a single value). PostgreSQL also allows you to define indexes that cover index values. In fact, you can create an index that covers the entire array or (starting with PostgreSQL version 7.4) an index that covers individual elements within an array. In Chapter 2 we showed you a modified version of the customers table that included an array column (monthly_balances). You can add this column to your working copy of the customers table with the following command:

```
movies=#  ALTER TABLE customers
movies-#    ADD COLUMN
movies-#      monthly_balances DECIMAL( 7, 2 )[ 12 ];
ALTER TABLE
```

To create an index that covers a single element of `monthly_balances` array (say, the element corresponding to the month of February), you could execute the following command:

```
movies=# CREATE INDEX customers_feb
movies-#   ON customers(( monthly_balances[2] ));
CREATE INDEX
```

Notice that you need an extra set of parentheses around `monthly_balances[2]`. Once you've created the `customers_feb` index, PostgreSQL can use it to satisfy queries such as

```
movies=# SELECT * FROM customers WHERE monthly_balances[2] = 10;
movies=# SELECT * FROM customers ORDER BY monthly_balances[2];
```

To create an index that covers the entire `monthly_balances` array, execute the command

```
movies=# CREATE INDEX customers_by_monthly_balance
movies-#   ON customers( monthly_balances );
CREATE INDEX
```

When you create an index that covers an array column, the syntax is the same as you would use to cover a scalar (single-valued) column. The PostgreSQL optimizer can use the `customers_by_monthly_balance` index to satisfy an `ORDER BY` clause such as

```
movies=# SELECT * FROM customers ORDER BY monthly_balances;
```

However, you may be surprised to find that the optimizer will *not* use `customers_by_monthly_balance` to satisfy a `WHERE CLAUSE` such as

```
movies=# SELECT * FROM customers WHERE monthly_balances[1] = 10;
```

The PostgreSQL optimizer *will* use the `customers_by_monthly_balance` index to satisfy a `WHERE_CLAUSE` that compares the entire `monthly_balances` array against another array, like this:

```
movies=# SELECT * FROM customers WHERE monthly_balances = '{10}';
```

But be aware that these queries are not equivalent. The first `WHERE` clause (`monthly_balances[1] = 10`) selects any row where `monthly_balances[1]` is equal to `10`, regardless of the other `monthly_balances` in that row. The second `WHERE` clause (`monthly_balances = '{10}'`) selects only those rows where `monthly_balances[1] = 10` and all other `monthly_balances` values are `NULL`.

Indexes and Tablespaces

When you create an index, you can tell PostgreSQL to store the index in a specific tablespace by including a `TABLESPACE` *`tablespacename`* clause, like this:

```
CREATE INDEX rentals_rental_date
    ON rentals ( rental_date ) TABLESPACE mytablespace;
```

If you don't specify a tablespace, PostgreSQL creates the index in the tablespace assigned to the table that you are indexing. You can move an existing index to a different tablespace using the `ALTER INDEX` command. For example, to move the `rentals_rental_date` index to `mytablespace`, you would execute the command

```
ALTER INDEX rentals_rental_date SET TABLESPACE mytablespace;
```

You may want to store a table and its indexes in different tablespaces in order to spread the workload among multiple physical disk drives.

Getting Information About Databases and Tables

When you create a table, PostgreSQL stores the definition of that table in the system catalog. The system catalog is a collection of PostgreSQL tables. You can issue `SELECT` statements against the system catalog tables just like any other table, but there are easier ways to view table and index definitions.

When you are using the psql client application, you can view the list of tables defined in your database using the `\d` meta-command:

```
movies=# \d
            List of relations
       Name       | Type  |     Owner
------------------+-------+---------------
 customers        | table | bruce
 rentals          | table | bruce
 tapes            | table | bruce
```

To see the detailed definition of a particular table, use the `\d` *`table-name`* meta-command:

```
movies=# \d tapes
                Table "tapes"
  Column  |         Type          | Modifiers
----------+-----------------------+-----------
 tape_id  | character(8)          | not null
 title    | character varying(80) | not null
 duration | interval              |
```

You can also view a list of all indexes defined in your database. The `\di` meta-command displays indexes:

```
movies=# \di
                   List of relations
 Schema |           Name            | Type  | Owner |   Table
--------+---------------------------+-------+-------+-----------
 public | customers_customer_id_key | index | korry | customers
```

You can see the full definition for any given index using the `\d` *`index-name`* meta-command:

```
movies=# \d customers_customer_id_key
Index "public.customers_customer_id_key"
   Column    |  Type
-------------+---------
 customer_id | integer
UNIQUE, btree, for table "public.customers"
```

Table 3.1 shows a complete list of the system catalog-related meta-commands in `psql`:

Table 3.1 **System Catalog Meta-Commands**

Command	Result
`\dd` *`object-name`*	Display comments for *object-name*
`\db`	List all tablespaces
`\dn`	List all schemas
`\d_\dt`	List all tables
`\di`	List all indexes
`\ds`	List all sequences
`\dv`	List all views
`\dS`	List all PostgreSQL-defined tables
`\d table-name`	Show table definition
`\d index-name`	Show index definition
`\d view-name`	Show view definition
`\d sequence-name`	Show sequence definition
`\dp`	List all privileges
`\dl`	List all large objects
`\da`	List all aggregates
`\df`	List all functions
`\dc`	List all conversions
`\dC`	List all casts
`\df function-name`	List all functions with given name
`\do`	List all operators
`\do operator-name`	List all operators with given name
`\dT`	List all types
`\dD`	List all domains
`\dg`	List all groups
`\du`	List all users
`\l`	List all databases in this cluster

Alternative Views (Oracle-Style Dictionary Views)

One of the nice things about an open-source product is that code contributions come from many different places. One such project exists to add Oracle-style dictionary views to PostgreSQL. If you are an experienced Oracle user, you will appreciate this feature. The orapgsqlviews project contributes Oracle-style views such as `all_views`, `all_tables`, `user_tables`, and so on. For more information, see http://gborg.postgresql.org.

PostgreSQL version 8.0 introduced a set of views known as the `INFORMATION_SCHEMA`. The views defined in the `INFORMATION_SCHEMA` give you access to the information stored in the PostgreSQL system tables. The `INFORMATION_SCHEMA` is defined as part of the SQL standard and you'll find an `INFORMATION_SCHEMA` in most commercial (and a few open-source) database systems. If you become familiar with the views defined in the `INFORMATION_SCHEMA`, you'll find it much easier to move from one RDBMS system to another—every `INFORMATION_SCHEMA` contains the same set of views, each containing the same set of columns. For example, to see a list of the tables defined in your current database, you could execute the command:

```
SELECT table_schema, table_name, table_type FROM information_schema.tables;
```

You can execute that same query in DB2, MS SQL Server, or Informix (sadly, Oracle doesn't support the `INFORMATION_SCHEMA` standard at the time we are writing this). So what can you find in the `INFORMATION_SCHEMA`?

- `schemata`—Lists the schemas (in the current database) that are owned by you
- `tables`—Lists all tables in the current database (actually, you only see those tables that you have the right to access in some way)
- `columns`—Lists all columns in all tables that you have the right to access
- `views`—Lists all of the views you have access to in the current database
- `table_privileges`—Shows the privileges you hold (or that you granted) for each accessible object in the current database
- `domains`—Lists all of the domains defined in the current database
- `check_constraints`—Lists all of the `CHECK` constraints defined for the accessible tables (or domains) in the current database

There are more views in the `INFORMATION_SCHEMA` than we've described here (in fact, there are a total of 39 `INFORMATION_SCHEMA` views in PostgreSQL 8.0). See Chapter 30, "The Information Schema," of the PostgreSQL user guide for a complete list.

Why would you want to use the `INFORMATION_SCHEMA` instead of psql's `\d` commands? We can think of three reasons. First, you can use the `INFORMATION_SCHEMA` inside of your own client applications—you can't do that with the `\d` commands because they are part of the psql console application (itself a PostgreSQL client) instead of the PostgreSQL server. Second, by using the views defined in the `INFORMATION_SCHEMA`, you can read the PostgreSQL system tables using the same queries that you would use to read the DB2 system tables (or Sybase or SQL Server). That makes your client applications a bit more portable. Finally, you can write *custom queries* against the views defined

in the INFORMATION_SCHEMA—you can't customize the \d commands. For example, if you need to find all of the date columns in your database, just look inside of INFORMATION_SCHEMA.columns, like this:

```
SELECT DISTINCT table_name
    FROM information_schema.columns WHERE data_type = 'date';
```

Need to know which columns can hold a NUMERIC value of at least seven digits? Use this query:

```
SELECT table_name,column_name, numeric_precision
    FROM information_schema.columns
    WHERE data_type = 'numeric' AND numeric_precision >= 7;
```

Of course, you can find all the information exposed by the INFORMATION_SCHEMA in the PostgreSQL system tables (pg_class, pg_index, and so on), but the INFORMATION_SCHEMA is often much easier to work with. The INFORMATION_SCHEMA views usually contain human-readable names for things like data type names, table names, and so on—the PostgreSQL system tables typically contain OIDs that you have to JOIN to another table in order to come up with a human-readable name.

Transaction Processing

Now let's move on to an important feature in any database system: transaction processing.

A *transaction* is a group of one or more SQL commands treated as a unit. PostgreSQL promises that all commands within a transaction will complete or that none of them will complete. If any command within a transaction does not complete, PostgreSQL will roll back all changes made within the transaction.

PostgreSQL makes use of transactions to ensure database consistency. Transactions are needed to coordinate updates made by two or more concurrent users. Changes made by a transaction are not visible to other users until the transaction is *committed*. When you commit a transaction, you are telling PostgreSQL that all the changes made within the transaction are logically complete, the changes should be made permanent, and the changes should be exposed to other users. When you roll back a transaction, you are telling PostgreSQL that the changes made within the transaction should be discarded and not made visible to other users.

To start a new transaction, execute a BEGIN command. To complete the transaction and have PostgreSQL make your changes permanent, execute the COMMIT command. If you want PostgreSQL to revert all changes made within the current transaction, execute the ROLLBACK command[11].

It's important to realize that *all* SQL commands execute within a transaction. If you don't explicitly BEGIN a transaction, PostgreSQL will automatically execute each command within its own transaction.

[11] BEGIN can also be written as BEGIN WORK or BEGIN TRANSACTION. COMMIT can also be written as COMMIT WORK or COMMIT TRANSACTION. ROLLBACK can also written as ROLLBACK WORK or ROLLBACK TRANSACTION.

Persistence

I used to think that single-command transactions were pretty useless: I was wrong. Single-command transactions are important because a single command can access multiple rows. Consider the following: Let's add a new constraint to the `customers` table.

```
movies=# ALTER TABLE customers ADD CONSTRAINT
movies-#   balance_exceeded CHECK( balance <= 50 );
```

This constraint ensures that no customer is allowed to have a balance exceeding $50.00. Just to prove that it works, let's try setting a customer's balance to some value greater than $50.00:

```
movies=# UPDATE CUSTOMERS SET balance = 100 where customer_id = 1;
ERROR:  ExecReplace: rejected due to CHECK constraint balance_exceeded
```

You can see that the `UPDATE` is rejected. What happens if you try to update more than one row? First, let's look at the data already in the `customers` table:

```
movies=# SELECT * FROM customers;
 customer_id |    customer_name     |  phone   | birth_date | balance
-------------+----------------------+----------+------------+---------
           1 | Jones, Henry         | 555-1212 | 1970-10-10 |    0.00
           2 | Rubin, William       | 555-2211 | 1972-07-10 |   15.00
           3 | Panky, Henry         | 555-1221 | 1968-01-21 |    0.00
           4 | Wonderland, Alice N. | 555-1122 | 1969-03-05 |    3.00
           8 | Wink Wankel          | 555-1000 | 1988-12-25 |    0.00
(5 rows)
```

Now, try to `UPDATE` every row in this table:

```
movies=# UPDATE customers SET balance = balance + 40;
ERROR:  ExecReplace: rejected due to CHECK constraint balance_exceeded
```

This `UPDATE` command is rejected because adding $40.00 to the balance for `Rubin, William` violates the `balance_exceeded` constraint. The question is, were any of the `customers` updated before the error occurred? The answer is: probably. You don't really know for sure because any changes made before the error occurred are rolled back. The net effect is that no changes were made to the database:

```
movies=# SELECT * FROM customers;
 customer_id |    customer_name     |  phone   | birth_date | balance
-------------+----------------------+----------+------------+---------
           1 | Jones, Henry         | 555-1212 | 1970-10-10 |    0.00
           2 | Rubin, William       | 555-2211 | 1972-07-10 |   15.00
           3 | Panky, Henry         | 555-1221 | 1968-01-21 |    0.00
           4 | Wonderland, Alice N. | 555-1122 | 1969-03-05 |    3.00
           8 | Wink Wankel          | 555-1000 | 1988-12-25 |    0.00
(5 rows)
```

If some of the changes persisted while others did not, you would have to somehow find the persistent changes yourself and revert them. You can see that single-command transactions are far from useless. It took me awhile to learn that lesson.

What about multicommand transactions? PostgreSQL treats a multicommand transaction in much the same way that it treats a single-command transaction. A transaction is *atomic*, meaning that all the commands within the transaction are treated as a single unit. If any of the commands fail to complete, PostgreSQL reverts the changes made by other commands within the transaction.

Transaction Isolation

I mentioned earlier in this section that the changes made within a transaction are not visible to other users until the transaction is committed. To be a bit more precise, uncommitted changes made in one transaction are not visible to other transactions[12].

Transaction isolation helps to ensure consistent data within a database. Let's look at a few of the problems solved by transaction isolation.

Consider the following transactions:

User: bruce	Time	User: sheila
`BEGIN TRANSACTION`	T1	`BEGIN TRANSACTION`
`UPDATE customers`	T2	
`  SET balance = balance - 3`		
`  WHERE customer_id = 2;`		
	T3	`SELECT SUM( balance )`
		`  FROM customers;`
	T4	`COMMIT TRANSACTION;`
`ROLLBACK TRANSACTION;`	T5	

At time T1, `bruce` and `sheila` each begin a new transaction. `bruce` updates the balance for customer 3 at time T1. At time T3, `sheila` computes the `SUM()` of the balances for all customers, completing her transaction at time T4. At time T5, `bruce` rolls back his transaction, discarding all changes within his transaction. If these transactions were not isolated from each other, `sheila` would have an incorrect answer: Her answer was calculated using data that was rolled back.

This problem is known as the *dirty read* problem: without transaction isolation, `sheila` would read uncommitted data. The solution to this problem is known as `READ COMMITTED`. `READ COMMITTED` is one of the two transaction isolation levels supported by

[12] This distinction is important when using (or developing) a client that opens two or more connections to the same database. Transactions are not shared between multiple connections. If you make an uncommitted change using one connection, those changes will not be visible to the other connection (until committed).

PostgreSQL. A transaction running at the `READ COMMITTED` isolation level is not allowed to read uncommitted data. I'll show you how to change transaction levels in a moment.

There are other data consistency problems that are avoided by isolating transactions from each other. In the following scenario, `sheila` will receive two different answers within the same transaction:

User: bruce	Time	User: sheila
`BEGIN TRANSACTION;`	T1	`BEGIN TRANSACTION;`
	T2	`SELECT balance FROM customers WHERE customer_id = 2;`
`UPDATE customers SET balance = 20 WHERE customer_id = 2;`	T3	
`COMMIT TRANSACTION;`	T4	
	T5	`SELECT balance FROM customers WHERE customer_id = 2;`
	T6	`COMMIT TRANSACTION;`

Again, `bruce` and `sheila` each start a transaction at time T1. At T2, `sheila` finds that customer 2 has a balance of $15.00. `bruce` changes the balance for customer 2 from $15.00 to $20.00 at time T3 and commits his change at time T4. At time T5, `sheila` executes the same query that she executed earlier in the transaction, but this time she finds that the balance is $20.00. In some applications, this isn't a problem; in others, this interference between the two transactions is unacceptable. This problem is known as the *non-repeatable read*.

Here is another type of problem:

User: bruce	Time	User: sheila
`BEGIN TRANSACTION;`	T1	`BEGIN TRANSACTION;`
	T2	`SELECT * FROM customers;`
`INSERT INTO customers VALUES ( 6, 'Neville, Robert', '555-9999', '1971-03-20', 0.00 );`	T3	

User: bruce	Time	User: sheila
`COMMIT TRANSACTION;`	T4	
	T5	`SELECT * FROM customers;`
	T6	`COMMIT TRANSACTION;`

In this example, `sheila` again executes the same query twice within a single transaction. This time, `bruce` has inserted a new row in between the `sheila`'s queries. Notice that this is not a case of a *dirty read*—`bruce` has committed his change before `sheila` executes her second query. At time T5, `sheila` finds a new row. This is similar to the non-repeatable read, but this problem is known as the *phantom read* problem.

The answer to both the non-repeatable read and the phantom read is the `SERIALIZABLE` transaction isolation level. A transaction running at the `SERIALIZABLE` isolation level is only allowed to see data committed before the transaction began.

In PostgreSQL, transactions usually run at the `READ COMMITTED` isolation level. If you need to avoid the problems present in `READ COMMITTED`, you can change isolation levels using the `SET TRANSACTION` command. The syntax for the `SET TRANSACTION` command is

```
SET TRANSACTION ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE };
```

The `SET TRANSACTION` command affects only the current transaction (and it must be executed before the first DML[13] command within the transaction). If you want to change the isolation level for your session (that is, change the isolation level for future transactions), you can use the `SET SESSION` command:

```
SET SESSION CHARACTERISTICS AS
    TRANSACTION ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE }
```

PostgreSQL version 8.0 introduces a new transaction processing feature called a `SAVEPOINT`. A `SAVEPOINT` is a named marker that you define within the stream of commands that make up a transaction. Once you've defined a `SAVEPOINT`, you can `ROLLBACK` any changes that you've made since that point without discarding changes made prior to the `SAVEPOINT`—in other words, you can `ROLLBACK` part of a transaction (the trailing part) without rolling back the entire transaction. To create a `SAVEPOINT`, execute a `SAVEPOINT` command within a transaction. The syntax for a `SAVEPOINT` command is very simple:

```
SAVEPOINT savepoint-name
```

The *savepoint-name* must follow the normal rules for an identifier; it must be unique within the first 64 characters and must start with a letter or underscore (or it must be a quoted identifier). A `SAVEPOINT` gives a name to a point in time; in particular, a point between two SQL commands. Consider the following sequence:

[13] A DML (data manipulation language) command is any command that can update or read the data within a table. `SELECT`, `INSERT`, `UPDATE`, `FETCH`, and `COPY` are DML commands.

```
movies=# SELECT customer_id, customer_name FROM customers;
 customer_id |    customer_name
-------------+---------------------
           3 | Panky, Henry
           1 | Jones, Henry
           4 | Wonderland, Alice N.
           2 | Rubin, William
(4 rows)

movies=# START TRANSACTION;
START TRANSACTION

movies=# INSERT INTO customers VALUES( 5, 'Kemp, Hans' );
INSERT 44272 1

movies=# SELECT * FROM customers;
 customer_id |    customer_name     |  phone   | birth_date | balance
-------------+----------------------+----------+------------+---------
           3 | Panky, Henry         | 555-1221 | 1968-01-21 |    0.00
           1 | Jones, Henry         | 555-1212 | 1970-10-10 |    0.00
           4 | Wonderland, Alice N. | 555-1122 | 1969-03-05 |    3.00
           2 | Rubin, William       | 555-2211 | 1972-07-10 |   15.00
           5 | Kemp, Hans           |          |            |
(5 rows)
```

At this point, you've started a new transaction and inserted a new row, but you haven't committed your changes yet. Now define a `SAVEPOINT` named `p1` and insert a second row:

```
movies=# SAVEPOINT P1;
SAVEPOINT

movies=# INSERT INTO customers VALUES( 6, 'Falkstein, Gerhard' );
INSERT 44273 1
```

The `SAVEPOINT` command inserted a marker into the transaction stream. If you execute a `ROLLBACK` command at this point, both of the newly inserted rows will be discarded (in other words, *all* of the changes you've made in this transaction will be rolled back):

```
movies=# ROLLBACK;
ROLLBACK
movies=# SELECT * FROM customers;
 customer_id |    customer_name     |  phone   | birth_date | balance
-------------+----------------------+----------+------------+---------
           3 | Panky, Henry         | 555-1221 | 1968-01-21 |    0.00
           1 | Jones, Henry         | 555-1212 | 1970-10-10 |    0.00
           4 | Wonderland, Alice N. | 555-1122 | 1969-03-05 |    3.00
           2 | Rubin, William       | 555-2211 | 1972-07-10 |   15.00
(4 rows)
```

Now repeat the same sequence of commands, but this time around, execute a qualified ROLLBACK command, like this:

```
movies=# ROLLBACK TO SAVEPOINT P1;
ROLLBACK
movies=# SELECT * FROM customers;
 customer_id |    customer_name     |  phone   | birth_date | balance
-------------+----------------------+----------+------------+---------
           3 | Panky, Henry         | 555-1221 | 1968-01-21 |    0.00
           1 | Jones, Henry         | 555-1212 | 1970-10-10 |    0.00
           4 | Wonderland, Alice N. | 555-1122 | 1969-03-05 |    3.00
           2 | Rubin, William       | 555-2211 | 1972-07-10 |   15.00
           5 | Kemp, Hans           |          |            |
(5 rows)
```

When you ROLLBACK to a SAVEPOINT, changes made since the SAVEPOINT are discarded, but not changes made before the SAVEPOINT. So, you see that the customers table retains the first row that you inserted, but not the second row. When you ROLLBACK to a SAVEPOINT, you are still in the middle of a transaction—you must complete the transaction with a COMMIT or ROLLBACK command.

Here are a few important points to keep in mind when you're working with SAVEPOINTs:

- You can nest SAVEPOINTs. For example, if you create a SAVEPOINT named P1, then create a second SAVEPOINT named P2, you have created a nested SAVEPOINT (P2 is nested within P1). If you ROLLBACK TO SAVEPOINT P2, PostgreSQL discards any changes made since P2, but preserves changes made between P1 and P2. On the other hand, if you ROLLBACK TO SAVEPOINT P1, PostgreSQL discards all changes made since P1, including all changes made since P2. Nested SAVEPOINTs are handy when you are working with a multilevel table structure such as ORDERS and LINEITEMS (where you have multiple line items per order). If you define a SAVEPOINT prior to modifying each order, and a second, nested SAVEPOINT prior to modifying each line item, you can ROLLBACK changes made to a single line item, changes made to a single order, or an entire transaction.
- You can use the same SAVEPOINT name as often as you like within a single transaction—the new SAVEPOINT simply replaces the old SAVEPOINT[14]. Again, this is useful when you are working with a multilevel table structure. If you create a SAVEPOINT prior to processing each line item and you give each of those SAVEPOINTs the same name, you can ROLLBACK changes made to the most recently processed line item.

[14] PostgreSQL doesn't follow the SQL standard when you create two SAVEPOINTs within the same transaction. PostgreSQL simply hides the old SAVEPOINT—the SQL standard states that the old SAVEPOINT should be destroyed. If you need the SQL-prescribed behavior, you can destroy the old SAVEPOINT with the command RELEASE SAVEPOINT *savepoint-name*.

- If you ROLLBACK to a SAVEPOINT, the SAVEPOINT is not destroyed—you can make more changes in the transaction and ROLLBACK to the SAVEPOINT again. However, any SAVEPOINTs nested within that SAVEPOINT will be destroyed. To continue the ORDERS and LINEITEMS example, if you ROLLBACK the changes made to an ORDERS row, you also discard changes made to the LINEITEMS for that ORDER and you are destroying the SAVEPOINT that you created for the most recent line item.
- If you make a mistake (such as a typing error), PostgreSQL rolls back to the most recent SAVEPOINT. That's a very nice feature. If you've used PostgreSQL for any length of time, you've surely exercised your vocabulary after watching PostgreSQL throw out a long and complex transaction because you made a simple typing error. If you insert SAVEPOINTs in your transaction, you won't lose as much work when your fingers fumble a table name.

Multi-Versioning and Locking

Most commercial (and open-source) databases use *locking* to coordinate multiuser updates. If you are modifying a table, that table is locked against updates and queries made by other users. Some databases perform page-level or row-level locking to reduce contention, but the principle is the same—other users must wait to read the data you have modified until you have committed your changes.

PostgreSQL uses a different model called *multi-versioning*, or *MVCC* for short (locks are still used, but much less frequently than you might expect). In a multi-versioning system, the database creates a new copy of the rows you have modified. Other users see the original values until you commit your changes—they don't have to wait until you finish. If you roll back a transaction, other users are not affected—they did not have access to your changes in the first place. If you commit your changes, the original rows are marked as obsolete and other transactions running at the READ COMMITTED isolation level will see your changes. Transactions running at the SERIALIZABLE isolation level will continue to see the original rows. Obsolete data is not automatically removed from a PostgreSQL database. It is hidden, but not removed. You can remove obsolete rows using the VACUUM command. The syntax of the VACUUM command is

```
VACUUM [ VERBOSE ] [ ANALYZE ] [ table ]
```

I'll talk about the VACUUM command in more detail in the next chapter.

The MVCC transaction model provides for much higher concurrency than most other models. Even though PostgreSQL uses multiple versions to isolate transactions, it is still necessary to lock data in some circumstances.

Try this experiment. Open two psql sessions, each connected to the movies database. In one session, enter the following commands:

```
movies=# BEGIN WORK;
BEGIN
movies=# INSERT INTO customers VALUES
movies-#  ( 5, 'Manyjars, John', '555-8000', '1960-04-02', 0 );
INSERT
```

In the other session, enter these commands:

```
movies=# BEGIN WORK;
BEGIN
movies=# INSERT INTO customers VALUES
movies-#  ( 6, 'Smallberries, John', '555-8001', '1960-04-02', 0 );
INSERT
```

When you press the `Enter` (or `Return`) key, this `INSERT` statement completes immediately. Now, enter this command into the second session:

```
movies=# INSERT INTO customers VALUES
movies-#  ( 5, 'Gomez, John', '555-8000', '1960-04-02', 0 );
```

This time, when you press `Enter`, `psql` hangs. What is it waiting for? Notice that in the first session, you already added a customer whose `customer_id` is 5, but you have not yet committed this change. In the second session, you are also trying to insert a customer whose `customer_id` is 5. You can't have two customers with the same `customer_id` (because you have defined the `customer_id` column to be the unique `PRIMARY KEY`). If you commit the first transaction, the second session would receive a *duplicate value* error. If you roll back the first transaction, the second insertion will continue (because there is no longer a constraint violation). PostgreSQL won't know which result to give you until the transaction completes in the first session.

Summary

Chapter 1, "Introduction to PostgreSQL and SQL," showed you some of the basics of retrieving and modifying data using PostgreSQL. In Chapter 2, "Working with Data in PostgreSQL," you learned about the many data types offered by PostgreSQL. This chapter has filled in some of the scaffolding—you've seen how to create new databases, new tables, and new indexes. You've also seen how PostgreSQL solves concurrency problems through its multi-versioning transaction model.

The next chapter, Chapter 4, "Performance," should help you understand how the PostgreSQL server decides on the fastest way to execute your SQL commands.